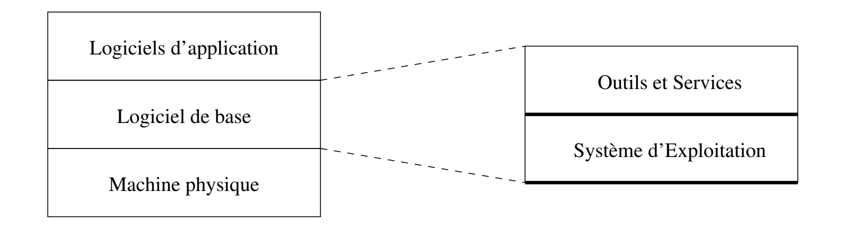
►► Chapitre 1 : Présentation générale ◄◄

- Notion de Système Informatique
- Les fonctions d'un système d'exploitation
- Historique et évolutions
- Structure interne d'un système d'exploitation
- Composantes d'un système d'exploitation

Notion de Système Informatique

Définition fonctionnelle :

- *Machine de traitement de l'information* qui assure
  - le stockage des données
  - l'exécution des programmes
- Structure en couches :



Les fonctions d'un système d'exploitation

Gestion de l'information :
- Structuration (codage, fichiers)
- Conservation (fichiers, mémoire)
- Transfert (E/S transparentes)
- Partage (entre plusieurs tâches)

Gestion des ressources :
- Allocation/Arbitrage
- Partage (diminution des coûts)
- Abstraction (simplification)

Autres services :
- Sécurité (traitement des erreurs)
- Évaluation/Statistique/Facturation
- Outils divers (sauvegarde, recherche)

►► Historique des S.E. ◄◄

Fin 40 : Organisation en *porte ouverte*.

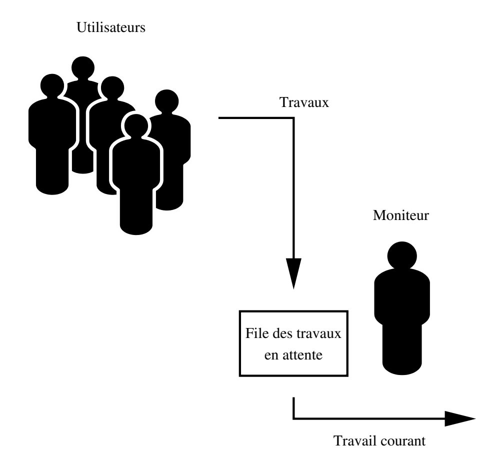
- Le temps d'accès à la machine est structuré en période de 30 minutes
- Chaque période est allouée à un utilisateur
- A la fin de la période la machine est réinitialisée
- L'utilisateur doit attendre la période suivante

Avantage : Chaque utilisateur a la machine pour lui tout seul.

Inconvénient : Le travail doit être découpé en tranche de 30 minutes.

Le moniteur humain de gestion des travaux

Apparition d'un opérateur de gestion des travaux :



Il récupère les travaux, gère la file d'attente et enchaine les exécutions.

Le moniteur logiciel d'enchaînement des travaux

Début 50 : *Moniteur logiciel d'enchaînement séquentiel des travaux*. Il assure les fonctions

– de compilation des travaux

– de chargement en mémoire

– d'enchainement des travaux

```
| chargement du moniteur compilateur
| compilation en mémoire du travail 1
| exécution du travail 1
| chargement du moniteur compilateur
| compilation en mémoire du travail 2
| exécution du travail 2
| ...
```


Le moniteur logiciel résidant

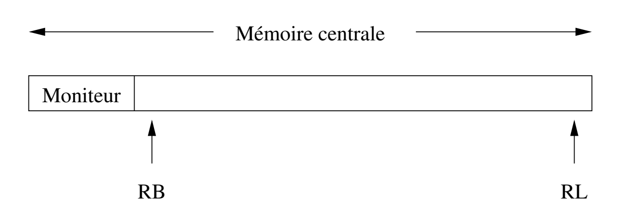
Milieux des années 50 : *Moniteur logiciel résidant d'enchaînement séquentiel des travaux*. Il assure les fonctions

– d'enchaînement automatique des travaux

– de protection de la mémoire

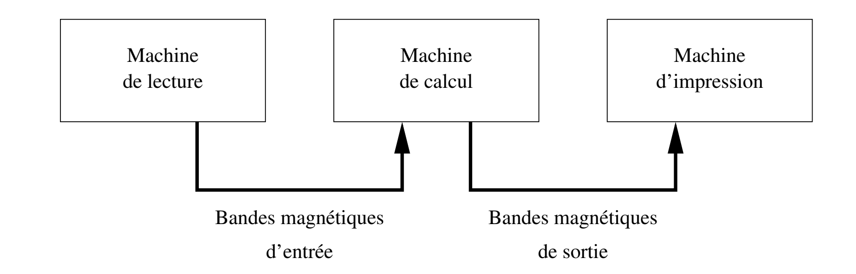
– de limitation de durée

– de supervision des entrées/sorties



Le traitement par lots

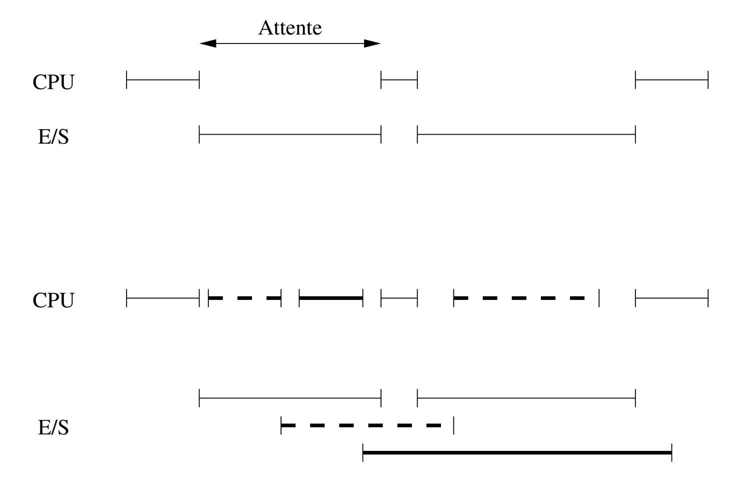
Fin des années 50 : *Traitement par lots*. Apparition du parallélisme des tâches entre lecture, calcul et impression.



ML	MC	MI
Préparation du lot 1		
Préparation du lot 2	Exécution du lot 1	
Préparation du lot 3	Exécution du lot 2	Impression du lot 1

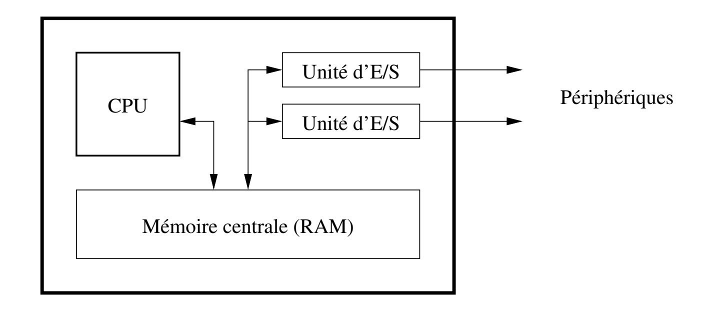
Cycles de CPU et d'entrée/sortie

Chaque processus enchaîne des **cycles de CPU** (exécution de code) et des **cycles d'entrée/sortie** :



Multiprogrammation

Début 60 : *Multiprogrammation*. Présence **simultanée** de **plusieurs programmes** en mémoire centrale.



Nouvelles caractéristiques :

- E/S tamponnées : définition d'un canal d'E/S,
- réimplantation du code,
- protection de la mémoire

Apparition des terminaux

60/70 : Intégration des terminaux :

– les terminaux sont connectés au serveur

– les utilisateurs travaillent devant le terminal

– ils partagent leur temps entre réflexion et action

Hypothèse 1 : Le temps de réflexion est de 90% :

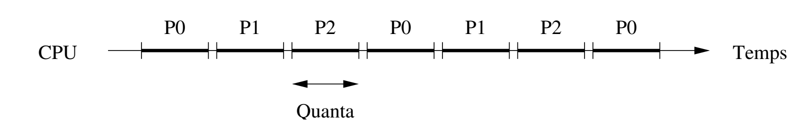
donc, sur 100 utilisateurs, **10 sont actifs**.

Hypothèse 2 : Les utilisateurs actifs réclament des actions simples :

prise en compte du travail interactif

60/70 : Le temps partagé

Le temps d'exécution de la CPU est découpé en tranches :



Si

Quanta = 50 millisecondes **et** une requête $\leq$ 1 quanta

alors

$$\text{Temps de réponse} = 10 \times 50 \text{ ms} = \frac{1}{2} \text{ s}$$

Contraintes :

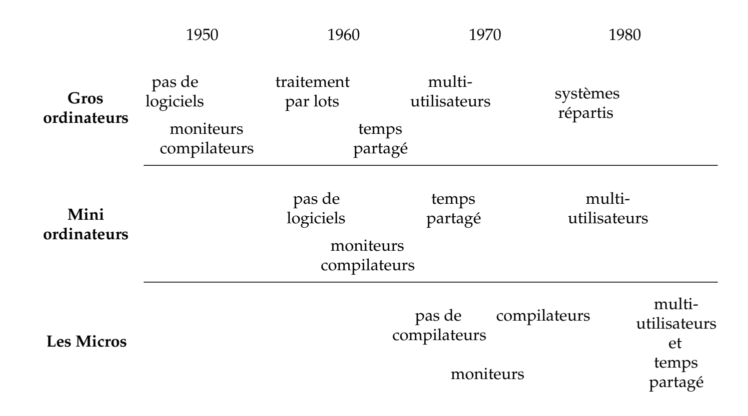
- multiprogrammation,
- temps de commutation faible,
- possibilité d'interruption « propre ».

Les systèmes répartis

1980 : *Systèmes répartis*. Répartition d'une activité entre plusieurs machines reliées par réseau (lent ou rapide).

Exemple : Le *Cluster*. La somme de plusieurs machines est vue comme une seule machine (simplicité, évolutivité, robustesse).

Un résumé de l'historique

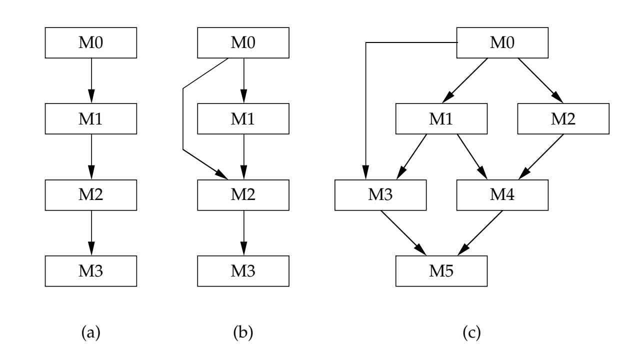


►► Structure interne d'un système ◄◄

Organisation en machines

Une *machine* c'est
- des objets,
- des actions possibles,
- des règles de composition des actions.

Dans une *structure en couches*, le S.E. est concu comme un empilement de machines.



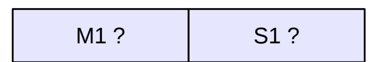
Avantages : Sécurité et modularité.

Utilité des machines

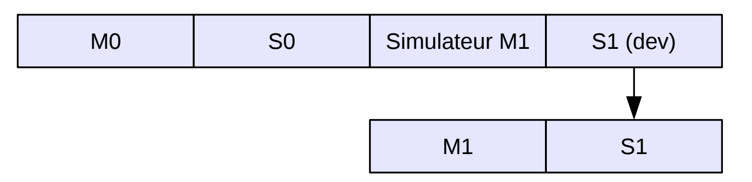
Ancienne machine M0 et son système S0 :

M0 | S0

On souhaite développer un système S1 sur une nouvelle machine M1 :

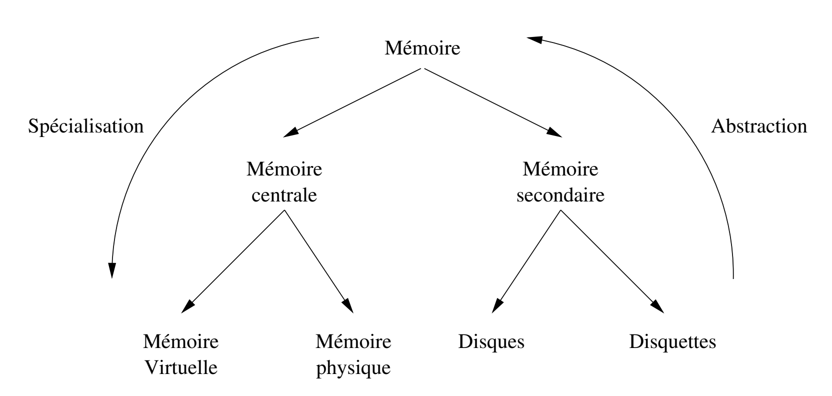


Un simulateur de M1 sur M0 est utilisé :



Organisation objet

Un *objet* c'est
- une représentation interne / externe,
- des fonctions d'accès,



Avantages : Sécurité, modularité, abstraction et spécialisation.

Un premier exemple

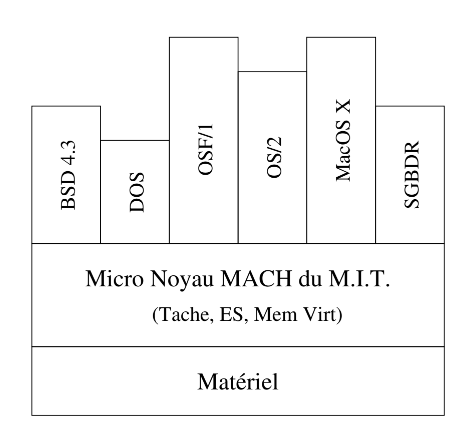
Les machines virtuelles d'IBM (VM)/CMS :

Utilisateurs
Noyau

Utilisateurs | Utilisateurs | Utilisateurs | Utilisateurs | Utilisateurs
Noyau | Noyau | Noyau | Noyau | Noyau
Machine virtuelle
Matériel

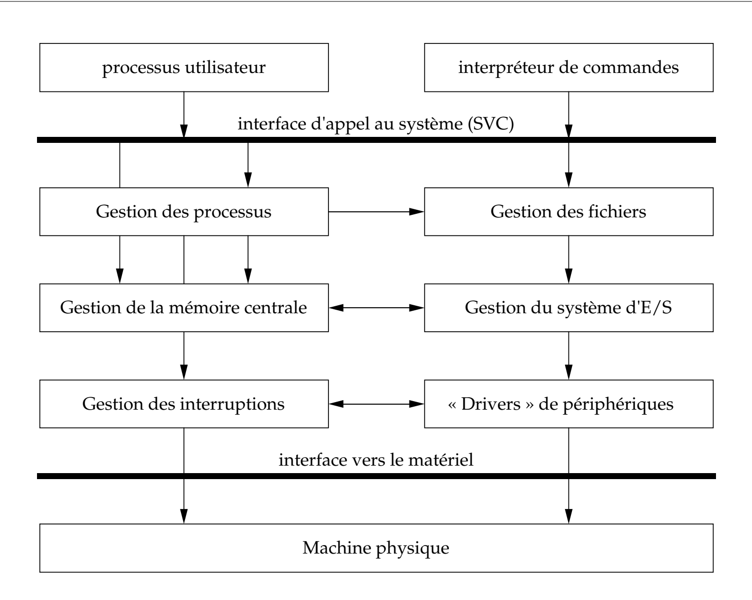
Un deuxième exemple

L'architecture « Micro-noyau » par opposition aux structures monolithiques des systèmes



Avantages : **Portabilité**, **Qualité**.

Composantes d'un S.E.



►► Présentation des processus ◄◄

- Notion de programmes
- Notion de machine
- Exécution d'un programme
- Processus

Notion de programme

Un **programme** est une suite d'instructions rangées en mémoire :

```
void main() {
  ...
}
```

⇒

2048	LOAD R1,
2052	$ 10
2056	INC R1
2060	ADD R1, R2

⇒

01001001011101101
10011111001010100
11101010001100101

Instructions =
- privilégiées =
  - Lancement des E/S
  - Contrôle des interruptions
  - Contrôle du matériel
- ordinaires =
  - mouvement de données
  - structure de contrôle
  - calcul
  - appel du système d'exploitation

Notion de machine

Une machine =
- Processeur (CPU)
- Mémoire
- Organes d'E/S =
  - Contrôleurs
  - Canaux

L'**état d'une machine** c'est l'état de ses composants.

Description du processeur

Un processeur =
- Mot d'état processeur (MEP) =
  - Registres spécialisés
  - Registres généraux =
    - Entier
    - Flottant
    - Adresse
- Unité de calcul (UAL)

Registres spécialisés =
- compteur ordinal (CO)
- mode d'exécution (MODE)
- pointeur de pile (SP)
- masque d'interruptions
- codes de conditions
- ...

- en mode **esclave** les instructions privilégiées sont interdites
- en mode **maître** les restrictions disparaissent

Exécution d'un programme

Une **exécution** c'est une évolution **discrète** de l'état de la machine.

Evolution des registres de la C.P.U.

Code en cours d'exécution

123	LOAD R1,
124	#10
125	INC R1
126	ADD R1, R2

CO	R1	R2
123	-	34

LOAD R1, #10

CO	R1	R2
125	10	34

INC R1

CO	R1	R2
126	11	34

ADD R1, R2

CO	R1	R2
127	45	34

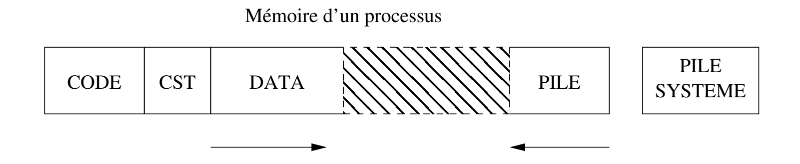
Exécution

Notion de points **observables**.

Notion de processus

Un **processus** est un programme en cours d'exécution. Il est défini par :

– des zones mémoire (code, pile et données),



– la valeur des registres de la CPU (un MEP),

– un ensemble de ressources allouées,

Le tout forme le **contexte d'exécution** du processus.

►► Les interruptions ◄◄

– Rendez-vous asynchrone,

– Interruption d'un processus,

– Interruption matérielle,

– Les appels système,

– Les déroutements.

► Rendez-vous asynchrones ◄

Problème : Prise en compte rapide d'un événement tout en effectuant un travail.

Exemple :

– *Le travail* :

- exécuter un processus consommateur de CPU (pas d'entrée/sortie)

– *Les évènements à gérer* :

- un paquet arrive sur la carte réseau
- déplacement de la souris
- activation du clavier
- apparition d'une alarme

► Attente active ◄

Solution : l'*attente active*. Soit F un indicateur qui signale l'événement

```
...
si (F = 1) alors
    F := 0
    < traiter l'événement >
fin si
...
```

– le test **explicite** est pénible et peu sécurisé

– perte de temps C.P.U.

► Attente active (suite) ◄

– pas de réaction immédiate

```
      Evt              Test
------|----------------|------
      |<--- latence --->|
```

– risque de collision

```
      Evt        Evt              Test
------|----------|----------------|------
      |<- écart ->|<--- latence --->|
```

écart minimum > latence

▶ Interruption d'un processus : Principe ◀

Interruption : opération **indivisible** effectuée par la C.P.U. qui change le `CO` et le `MODE`.

Les interruptions sont déclenchées **uniquement** sur les points observables (points **interruptibles**).

Chaque interruption a une cause identifiée par un entier.

Le **vecteur d'interruptions** (`VI`) est une table logée dans les adresses basses de la mémoire.

$$\text{interruption de cause } k = \begin{cases} \texttt{PUSH CO, MODE ;} \\ \texttt{MODE := maître ;} \\ \texttt{CO := vi[k] ;} \end{cases}$$

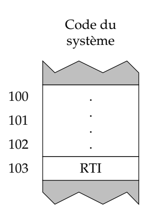
reprise après interruption :

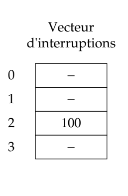
$$\text{instruction } \texttt{RTI} = \begin{cases} \texttt{POP MODE ;} \\ \texttt{POP CO} \end{cases}$$

▶ Interruption d'un processus : Exemple ◀

Code utilisateur

123	LOAD R1,
124	#10
125	INC R1
126	ADD R1, R2





Evolution des registres de la C.P.U.

CO	MODE	SP	R1	R2
123	esclave	200	-	34
LOAD R1, #10				
125	esclave	200	10	34
< interruption de cause n° 2 >				
100	maître	201	10	34
.	.	.		
103	maître	201	10	34
RTI				
125	esclave	200	10	34
INC R1				
126	esclave	200	11	34

Evolution de la pile

-	-	SP → première case
<125,e>	-	SP → deuxième case
<125,e>	-	SP → première case

▶ Structure générale d'un traitant ◀

Le code exécuté après une interruption est appelé le **traitant** de cette interruption.

Les traitants ont la structure suivante :

1. **sauvegarde** du contexte
2. **traitement** de la cause
3. **restauration** du contexte
4. retour au processus **interrompu** (instruction `RTI`)

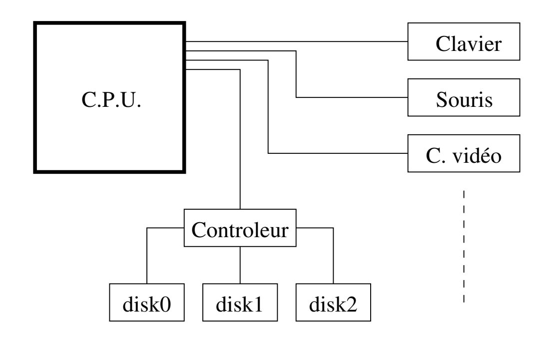
▶ Utilisation des interruptions ◀

Il existe trois types d'interruptions :

- **Interruption matérielle** : réaction aux événements extérieurs.
- **Appel au superviseur** : appel explicite d'une routine système.
- **Déroutement** : traitement des erreurs et des situations anormales.

▶ Interruption matérielle : principe ◀

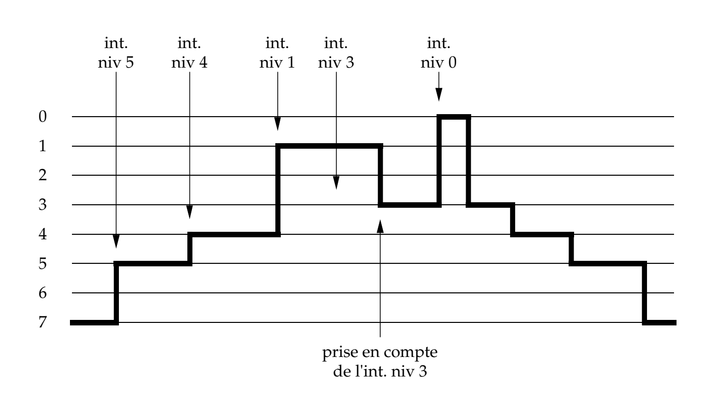
Un **signal électrique** est émis vers le processeur qui provoque une interruption.



Il y a donc un branchement vers le code de traitement des interruptions (**le traitant**) avec un **changement de mode**.

▶ Interruption matérielle : caractéristiques ◀

Le numéro associé à chaque interruption matérielle indique une **priorité** :



Des **instructions privilégiées** permettent de

- masquer des niveaux,
- armer ou désarmer des niveaux,
- déclencher des niveaux.

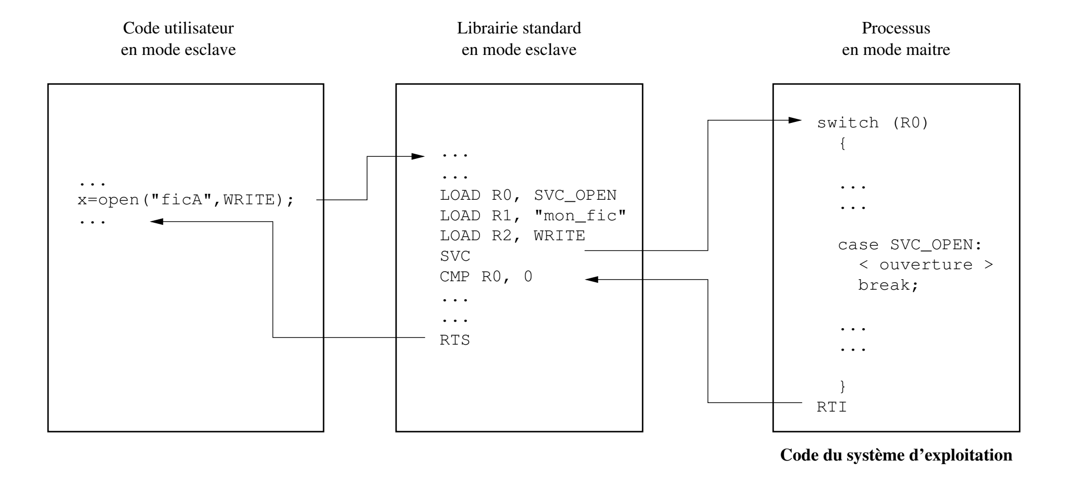
▶ Les appels systèmes : principe ◀

Si les processus s'exécutent en mode **utilisateur**, les entrées/sorties directes leurs sont interdits.

- ▶ Pour réaliser des E/S, les processus doivent envoyer des requêtes au système d'exploitation.
- ▶ L'appel de fonction classique pose des problèmes :
  - le nombre de points d'entrée est **important**
  - le mode d'exécution reste **esclave**
  - le code du système d'exploitation est accessible **en lecture**

► Les appels systèmes : moyens ◄

Les **trappes** (`Trap`) : Appel explicite du système par le biais d'une instruction qui **déclenche une interruption**.



► Les appels systèmes : Avantages ◄

Avantages des appels par interruption :

- Changement de **mode**
- Le nombre de points d'entrée est **limité**
- On obtient un **sas** qui isole le système d'exploitation
- Définition d'une **nouvelle machine** (ajout d'instructions)
- Une librairie standard offre
  - une interface système **agréable et simplifiée**
  - une interface **indépendante** du système d'exploitation
  - **l'implantation réelle** des appels systèmes en assembleur

► Les appels systèmes : Structure du traitant ◄

1. **sauvegarde** du contexte
2. vérifier la **conformité** de la requête
   - vérifier la nature de la requête
   - vérifier la nature des arguments
   - vérifier les droits du processus demandeur
3. **exécuter** la requête
4. **choisir** le prochain processus p à exécuter
5. **restaurer** le contexte de p
6. **relancer** le processus p

► Déroutements : Principe ◄

L'objectif des déroutements est double :

- traitement **systématique** des erreurs ou des situations anormales (défaut de page),
- **protection** et bonne utilisation de la machine (les processus s'exécutent sous surveillance).

Si l'exécution d'une instruction **produit une erreur**, alors
→ il y a **interruption** et
→ **branchement** vers le code de traitement des erreurs du système.

Les **causes** possibles sont :

- données incorrectes (division par zéro)
- opération interdite (instruction privilégiée)
- instruction inconnue
- accès à une zone mémoire interdite
- erreur de bus mémoire

▶ Déroutements : Exemples ◀

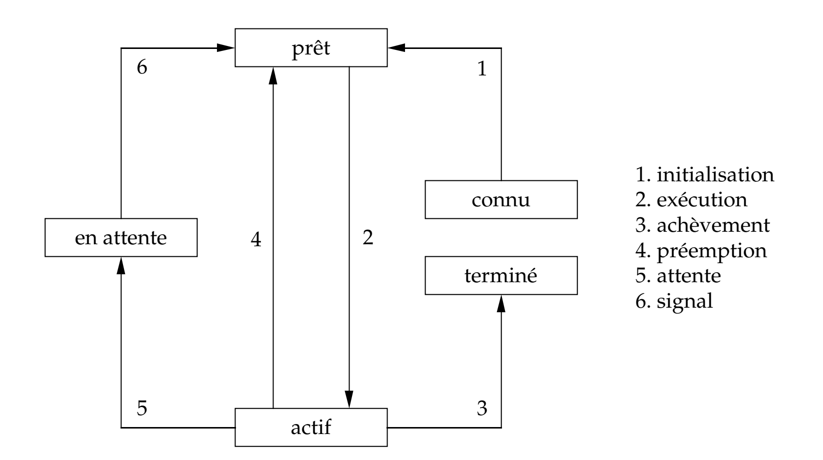
- Implantation d'une **gestion utilisateur des erreurs** (fonction `signal`)
- Gestion de la **mémoire virtuelle**.
- Ajout de **nouvelles instructions** par simulation logicielle.
- Le mode `TRACE` sur le processeur 68000 :
  - à chaque point interruptible, la C.P.U. génère un **déroutement**,
  - le système récupère donc le **contrôle** entre chaque instructions,
  - il peut visualiser **l'évolution des registres**.

►► Gestion des processus ◄◄

- État d'un processus,
- Représentation d'un processus,
- Rôle du PCB,
- Représentation des processus,
- Les « threads » (processus de poids léger),
- Utilité des « threads »,
- Implantation des « threads »,

► État d'un processus ◄

Chaque processus est dans l'un des **états opérationnels** suivants :



► Représentation d'un processus ◄

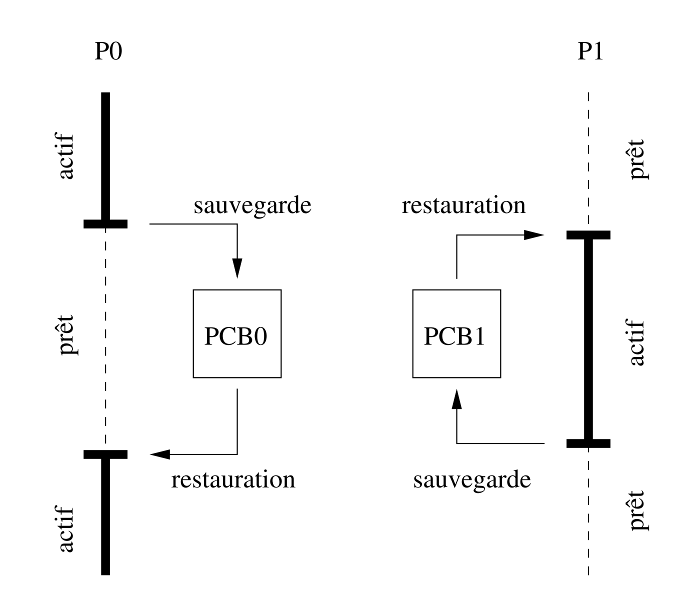
Pour chaque processus le système maintient :

- un identifiant (`pid`)
- un état opérationnel
- un contexte d'exécution
- des informations diverses :
  - priorités
  - filiations
  - propriétaire
- des statistiques :
  - temps CPU
  - nombre d'E/S
  - nombre de défauts de page

Ces informations sont rangées dans un PCB (**Process Control Block**).

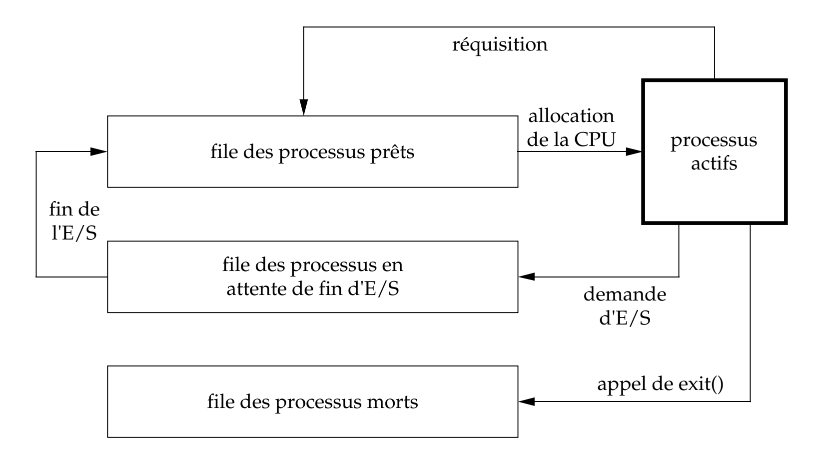
► Rôle du PCB ◄

Le rôle du PCB dans la commutation entre deux processus :



▶ Représentation des processus ◀

Les PCB sont rangés dans **des files** :



▶ Gestion des processus ◀

Création de processus :

- chaque processus peut **créer** des processus (ses fils),
- il est possible de **figer**, et de **reprendre** des processus.
- il est possible de **tuer** des processus ou demander leur arret.

Communication entre processus :

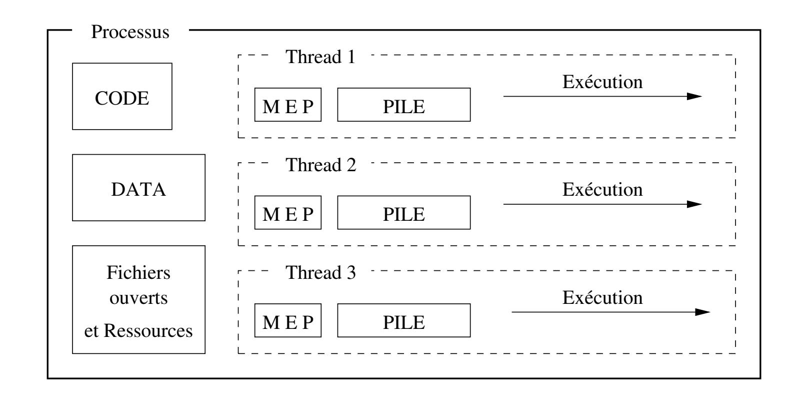
- prévoir des **outils de communication** (fichier, tube, socket, mémoire partagée, envoi de messages)

Synchronisation de processus :

- prévoir des **outils de synchronisation** (verrou, sémaphore, moniteur)

▶ Les « threads » (processus de poids léger) ◀

Un **thread** (**fil**) est un programme en cours d'exécution qui partage son code et ses données.



Chaque thread a une pile d'exécution **autonome**.

Un processus est composé de **threads**.

▶ Programmation des threads ◀

```
begin
   Instruction 1
   co-begin
      Instruction 2
      Instruction 3
   co-end
   Instruction 4
end
```

```
void travail() {
   int t ;
   instruction1() ;
   if ((t = thread()) == 0) {
      instruction3() ;
      thread_exit() ;
   }
   instruction2() ;
   thread_join(t) ;
   instruction4() ;
}
```

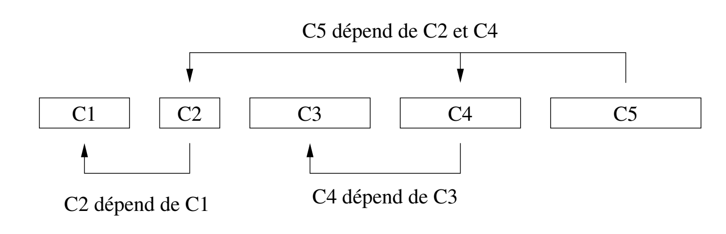
Trace de l'exécution :

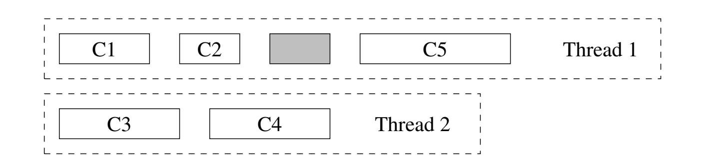
```
CPU0 :  |<--- Ins. 1 --->|<-- Ins. 2 -->|.....|<--- Ins. 4 --->

CPU1 :                   |<----- Ins. 3 ----->|
```

▶ Utilité des « threads » ◀

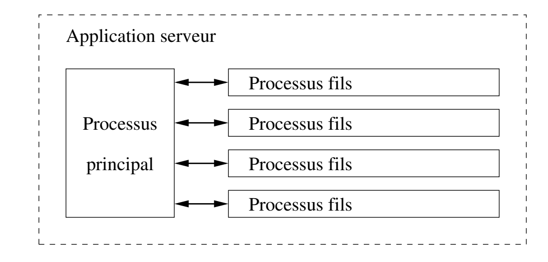
Étude des dépendances et découpage :

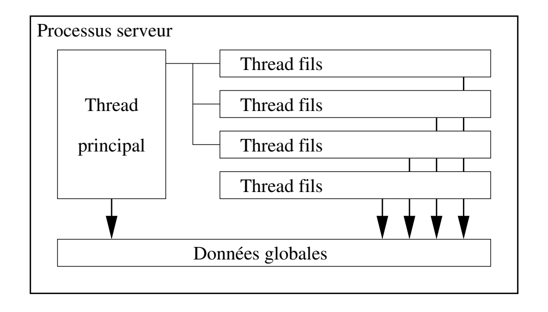




Avantages :
- récupération des temps d'E/S,
- exploitation des machines multi-processeurs,
- coopération entre threads.

▶ Utilité des « threads » pour les serveurs ◀

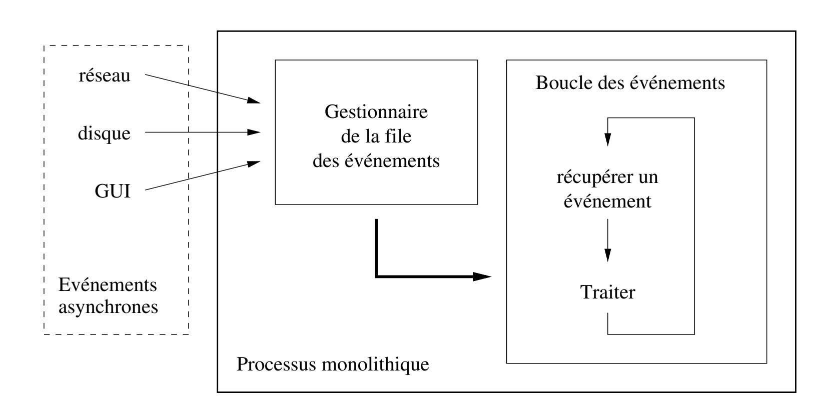




La **commutation** et la **communication** entre threads est une opération plus simple

▶ Les processus monolithiques ◀

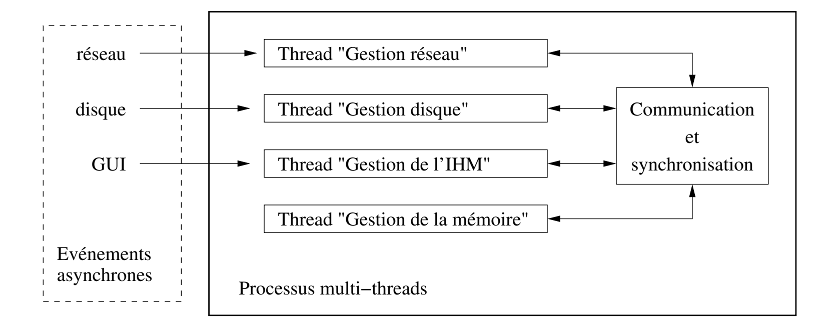
Il existe **un conflit** entre
- entrées / sorties synchrones (**bloquantes**),
- interface homme/machine (**IHM**).



Solution **d'attente active** :
- E/S asynchrones (**non-bloquantes**),
- structure avec **boucle d'événements**.

▶ Les processus multi-threads ◀

Il existe un **autre solution** basée sur
- le découpage en plusieurs **threads**,
- une utilisation des E/S **synchrones**,
- un module de **communication**.



Avantages :
- plus grande simplicité du code,
- indépendance entre les modules.

▶ Implantation des « threads » ◀

Implantation au niveau du S.E. :

+ le S.E. connaît et ordonnance les threads,
+ la répartition de la CPU est bonne,
– les structures du S.E. sont alourdies.

Implantation au niveau utilisateur (Java) :

+ une librairie se charge de la gestion des threads (création, ...),
+ la commutation entre threads d'un même processus est plus rapide,
– la répartition de la CPU n'est pas équitable,
– la mise « en attente » d'un processus entraîne le blocage de tous ses threads.

▶▶ Allocation du processeur ◀◀

– Allocation de la CPU
– Stratégies d'allocation
– Les algorithmes d'allocation :
  - FIFO
  - SJF
  - SRT
  - HRN
  - Le tourniquet
  - Le tourniquet multi-niveaux

▶ Allocation de la CPU ◀

Il y a perte de la CPU dans trois cas :

- interruption extérieur,
- déroutements,
- appels système.

Quel type d'allocation ?

– **le long terme** : variation du degré de multi-programmation par :
  - **swapping out** : les processus sont copiés en mémoire secondaire
  - **swapping in** : les processus sont rapatriés en mémoire centrale
– **le moyen terme** (détermine et classe les processus prêts),
– **le court terme** (répartition de la CPU)

▶ Stratégies d'allocation ◀

Objectifs :

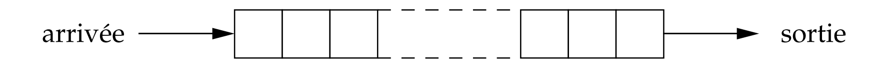
- équité,
- débit maximum
- maximum de processus interactifs,
- rationaliser la gestion des ressources,
- favoriser les « bons » processus,
- améliorer les temps de réponse moyen,

Critères :

- le taux d'E/S
- le taux d'utilisation de la CPU
- le type et la priorité du processus,
- temps CPU cumulé,
- temps CPU restant,
- taux de réquisition

▶ Algorithme d'allocation FIFO ◀

FIFO (**First In First Out**) : absence de réquisition et croissance rapide du temps de réponse.

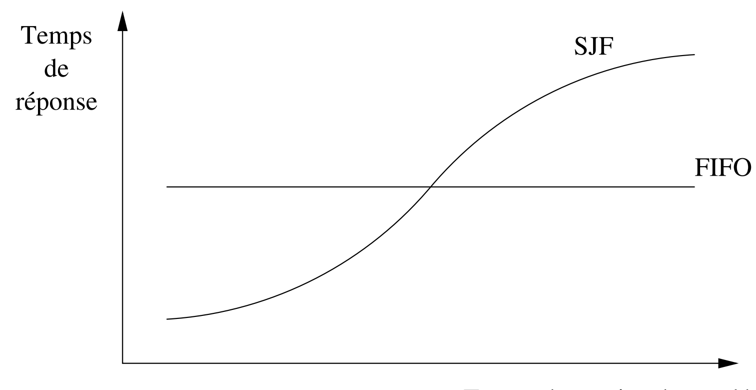


temps de réponse = (temps d'attente + temps d'exécution)

temps d'attente = (taille de la file FIFO × durée moyenne d'exécution)

▶ Algorithme d'allocation SJF ◀

SJF (**Short Job First**) : absence de réquisition et risque de privation.



Il faut estimer le temps d'exécution des processus.

▶ Algorithmes d'allocation SRT ◀

SRT (**Shortest Remaining Time**) : c'est SJF + un mécanisme de réquisition.

– Mêmes contraintes que SJF

– On choisit le processus qui a le temps restant d'exécution le plus petit

– L'arrivée d'un nouveau processus peut interrompre le processus courant

Conséquences :

- On favorise les travaux courts.
- Coûts d'exploitation importants.

▶ Algorithmes d'allocation HRN ◀

HRN (**Highest Response Ratio Next**) : Le choix est basé sur le ratio

$$p(t) = \frac{w(t) + t_s}{t_s}$$

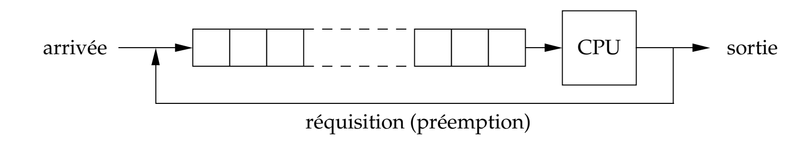
avec

- $w(t) = t - t_a$: temps d'attente,
- t_a : date d'arrivée,
- t_s : temps d'exécution estimé.

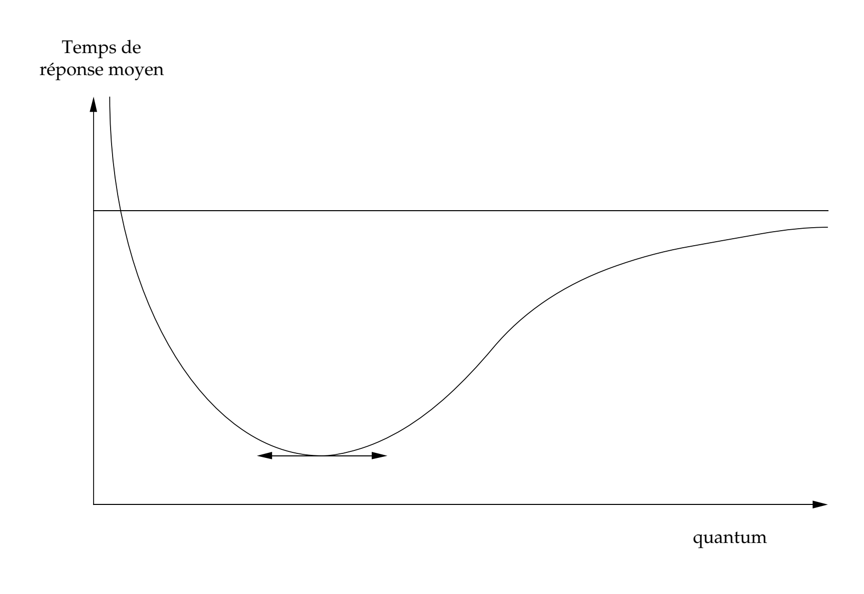
Si $w(t) = 0$, les travaux les plus courts (t_s) sont privilégiés.

▶ Algorithmes d'allocation du tourniquet ◀

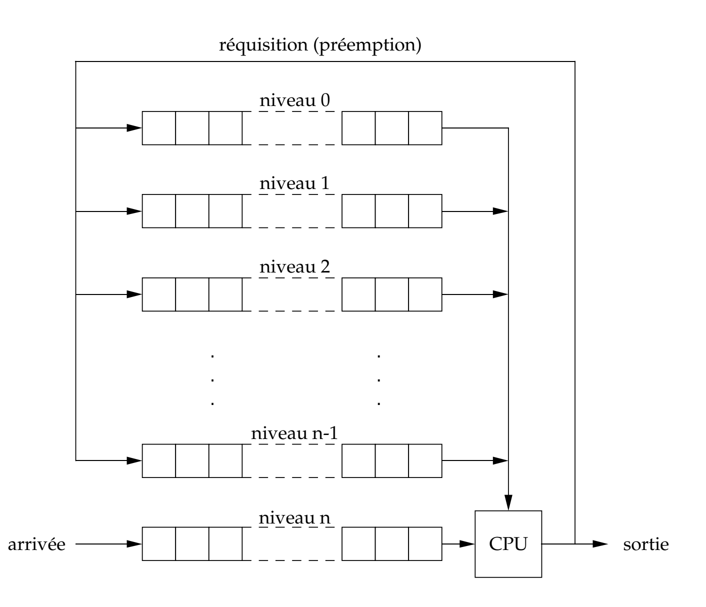
Tourniquet (**round robin**) : c'est FIFO plus un mécanisme de réquisition.



Comment fixer la durée du quanta :



▶ Tourniquet Multi-niveaux ◀



▶ Tourniquet Multi-niveaux avec priorités ◀

Il existe des classes de processus :

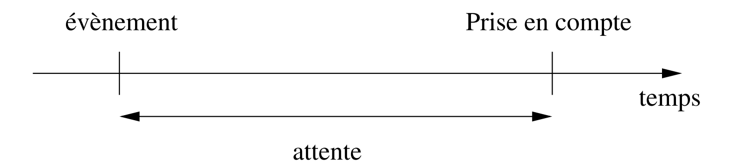
- processus systèmes,
- processus temps réel,
- processus interactifs,
- processus d'édition interactive,
- processus « batch ».

Chaque classe et **absolument** prioritaire sur celles de niveau inférieur.

A l'intérieur de chaque classe, les processus sont rangés dans un système de files multi-niveaux.

► Le cas des processus temps réel ◄

Prise en compte d'un évènement :



si

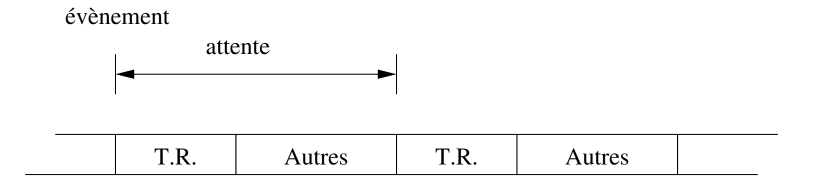
n est le nombre de processus temps réel,
s le temps pris par l'ordonnanceur,
t le temps moyen pris par les processus T.R.

alors

$$(n - 1) \times (s + t) < \text{attente}.$$

► Mélange Temps Réel et temps machine ◄

Le temps C.P.U. peut être divisé en une partie fixe réservée au processus Temps Réel, et une partie reservée aux autres processus.



Les algorithmes précédents s'appliquent dans la partie T.R.,

Les algorithmes classiques sont utilisés par la partie « Autres ».

► Allocation multi-processus ◄

Hypothèse : on dispose de N processeurs identiques.

Objectifs :

- équilibrage de la charge (homogénéité des taux d'utilisation de la C.P.U.),
- évitez les effets de « pompage »,

Système simple : le S.E. s'exécute toujours sur la même C.P.U.

Système plus sophistiqué : le S.E. est capable de s'exécuter en parallèle sur plusieurs processeurs.

→ Il faut gérer les problèmes de synchronisation.

▶▶ Présentation générale ◀◀

Le problème :

- Accès à des ressources partagées
- Les sections critiques
- Le problème de l'exclusion mutuelle

Solutions d'attente active :

- Une solution d'attente active
- blocage des interruptions
- Algorithme de Peterson (2 processus)
- Solution matérielle

Solutions de manipulation des processus :

- Les verrous
- Les sémaphores
- Le producteur et le consommateur
- Les sémaphores à messages
- Les régions critiques

Accès à des ressources partagées

Soit une pile **partagée** par plusieurs threads :

pile = **structure**
  sommet : **entier** ;
  data : **tableau** [1 .. Max] de **entier** ;

procédure empiler(**var** p : pile ; d : **entier**)
  p.sommet := p.sommet + 1 ;
  p.data[p.sommet] := d ;

Il est possible que nous ayons :

thread1 : empiler(p, 10)
thread1 : p.sommet := p.sommet + 1 ;
— — interruption — —
thread2 : empiler(p, 20)
thread2 : p.sommet := p.sommet + 1 ;
thread2 : p.data[p.sommet] := 20 ;
⋮ ⋮ ⋮
— — retour au thread 1 — —
thread1 : p.data[p.sommet] := 10 ;

Les sections critiques

Les ressources logicielles ou matérielles qui posent des problèmes sont dites **critiques**.

Les portions de code qui manipulent ces ressources critiques sont appelées des **sections critiques**.

Les notions de **ressource critique** et **section critique** sont utiles

– pour les threads d'un processus utilisateur,

– pour les données du système d'exploitation partagées par les processus.

Le problème de l'exclusion mutuelle

Forme des programmes :

⟨initialisation⟩ exécuté une seule fois

⋮

⟨prologue⟩
⟨**section critique**⟩
⟨épilogue⟩

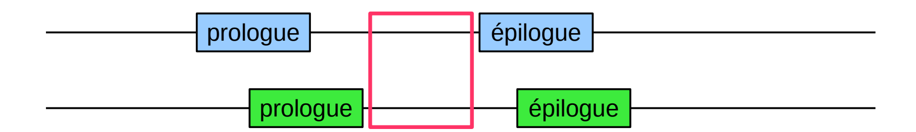
Contraintes :

– Il existe **au plus** un processus en section critique

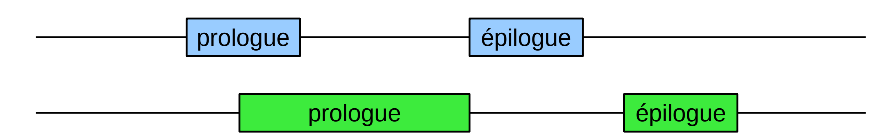
– Les processus ne sont pas bloqués sans raison (absence de **privation**).

– Les sections ⟨prologue⟩ et ⟨épilogue⟩ sont les mêmes pour tous les processus (**uniformité**).

Conflit avec deux threads. Le prologue et l'épilogue encadrent la section critique :



Le problème est réglé en augmentant la durée d'exécution du prologue :



Nouvelle formulation

La procédure empiler devient :

```
procédure empiler( var p : pile ; d : entier )
  ⟨prologue⟩
    p.sommet := p.sommet + 1 ;
    p.data[ p.sommet ] := d ;
  ⟨épilogue⟩
```

Nouvelle exécution :

```
thread1  :  empiler( p, 10 )
thread1  :  ⟨prologue⟩
thread1  :  p.sommet := p.sommet + 1 ;
        — — interruption — —
thread2  :  empiler( p, 20 )
thread2  :  ⟨prologue⟩
      — — blocage du thread 2 — —
   ⋮                    ⋮     ⋮
      — — retour au thread 1 — —
thread1  :  p.data[ p.sommet ] := 10 ;
thread1  :  ⟨épilogue⟩
   ⋮                    ⋮     ⋮
      — — reprise du thread 2 — —
thread2  :  p.sommet := p.sommet + 1 ;
thread2  :  p.data[ p.sommet ] := 20 ;
thread2  :  ⟨épilogue⟩
```


Une solution d'attente active

Soit *libre* une variable **partagée** de type booléenne pour coder l'exclusion mutuelle :

```
⟨init⟩       (1)  libre := vrai ;

                  ⋮

⟨prologue⟩   (2)  si (libre = faux) aller en (2)
             (3)  libre := faux

             (4)  ⟨section critique⟩

⟨épilogue⟩   (5)  libre := vrai
```

Critiques :

– Consommation de temps processeur

– L'exclusion mutuelle n'est pas toujours respectée :

2 ⟨interruption⟩ 2' 3' 4' ... ⟨retour au rouge⟩ 3 4 ...

blocage des interruptions

```
⟨init⟩       (1)  libre := vrai ;

⟨prologue⟩   (2)  soit p une variable privée
             (3)  répéter
             (4)     ⟨masquer les interruptions⟩
             (5)        p := libre ;
             (6)        libre := faux ;
             (7)     ⟨rétablir les interruptions⟩
             (8)  jusqu'à (p = vrai)
                       ⋮
                  ⟨section critique⟩
                       ⋮
⟨épilogue⟩   (9)  libre := vrai
```

Critiques :

- L'ordre d'arrivée n'est pas respecté.
- Risque de privation.
- C'est une solution **mono-processeur**.
- Utilisable uniquement en mode maître.

Solution pour deux processus

Soit *tour* une variable entière partagée :

var tour : **entier** ;

Voila le codage pour le processus P_i :

⟨initialisation⟩ (1) tour := 0 ;

⟨prologue⟩ (2) **répéter**
(3) **jusqu'à** (tour = i)

⟨épilogue⟩ (4) tour := $(1 - i)$;

Critiques :
- L'ordre est fixé à priori avec alternance
- Risque de privation (blocage inutile)
- Il n'y a pas de mémoire de l'état des processus

Algorithme de Peterson pour deux processus

On ajoute une variable *demande* pour représenter l'état des processus :

var tour : **entier** ;
demande : **tableau** [0 .. 1] de **booléen** ;

Voila le codage pour le processus P_i :

⟨initialisation⟩ (1) tour := 0 ;
(2) demande := (faux, faux) ;

⟨prologue⟩ (3) demande[i] := vrai ;
(4) tour := $1 - i$;
(5) **répéter**
(6) **jusqu'à** (tour = i) **ou** (demande[$1 - i$] = faux)

⟨épilogue⟩ (7) demande[i] := faux ;

C'est une solution correcte pour le problème de l'exclusion mutuelle à deux processus.

Solution matérielle

On introduit une nouvelle instruction *Test and Set* pour

– garantir l'atomicité d'une modification,

– construire une solution basée sur l'attente active valable en multi-processeurs :

Définition de *Test and Set* :

instruction TAS(**var** m : **entier** , **var** verrou : **entier**)
| ⟨bloquer la case mémoire verrou⟩
| | m := verrou
| | verrou := 0
| ⟨débloquer la case mémoire verrou⟩
| CO := CO + ⟨taille de l'instruction TAS⟩

Utilisation de Test and Set

Codage de l'exclusion mutuelle avec une variable partagée *mutex* et une variable privée *p*.

⟨initialisation⟩ (1) mutex := 1 ;

⟨prologue⟩ (2) **répéter**
(3) | TAS(p, mutex) ;
(4) **jusqu'à** (p = 1)

(5) ⟨section critique⟩

⟨épilogue⟩ (6) mutex := 1 ;

Critiques :
- Consommation de temps processeur.
- Risque de privation (cela peut s'arranger).
- Le processeur doit garantir l'exclusion mutuelle.

C'est une solution utilisable uniquement sur des **séquences brèves**.

Les verrous

Objectifs :

- ne plus perdre de temps CPU,
- simplicité de la solution.

Définition des verrous :

verrou : **structure**
  libre : booléen ;
  f : file FIFO de processus ;

procédure init(**var** *v* : verrou)
  v.libre := **vrai** ;
  v.f := {} ;

Un **verrou** est une structure de donnée **partagée** du système d'exploitation.

Les verrous (2)

Pour un verrou donné, les deux procédures ci-dessous s'exécutent en exclusion mutuelle.

procédure prendre(**var** *v* : verrou)
  si (*v.libre* = **faux**) **alors**
    ⟨soit P le processus appelant⟩
    ⟨entrer P dans la file *v.f*⟩
    ⟨suspendre le processus P⟩
  sinon
    v.libre := **faux** ;
  fin si

procédure libérer(**var** *v* : verrou)
  si ⟨la file *v.f* est vide⟩ **alors**
    v.libre := **vrai** ;
  sinon
    v.libre := **faux** ;
    ⟨sortir un processus Q de la file *v.f*⟩
    ⟨réveiller le processus Q⟩
  fin si

L'exclusion mutuelle avec les verrous

soit

var mutex : verrou ;

le code de l'exclusion mutuelle s'écrit

⟨initialisation⟩ (1) init(*mutex*) ;

⟨prologue⟩ (2) prendre(*mutex*) ;
  ⋮
  ⟨section critique⟩
  ⋮
⟨épilogue⟩ (3) libérer(*mutex*) ;

Difficultés des verrous

soit deux processus qui partagent deux ressources :

var *mutex1* : verrou ;
  mutex2 : verrou ;

  init(*mutex1*) ;
  init(*mutex2*) ;

⟨P_1⟩ (1) prendre(*mutex1*) ;
  (2)   prendre(*mutex2*) ;
  (3)   ⟨section critique⟩
  (4)   libérer(*mutex2*) ;
  (5) libérer(*mutex1*) ;

⟨P_2⟩ (1') prendre(*mutex2*) ;
  (2')   prendre(*mutex1*) ;
  (3')   ⟨section critique⟩
  (4')   libérer(*mutex1*) ;
  (5') libérer(*mutex2*) ;

Il y a blocage pour la séquence **1 . . . 1' . . . 2 3 . . . 3'**

Les sémaphores

Un verrou ne sait pas compter... donc, il faut remplacer le drapeau par un compteur (Dijkstra).

Un **sémaphore** est une structure de donnée **partagée** du système d'exploitation.

sémaphore : **structure**
- c : entier ;
- f : file FIFO de processus ;

procédure init(**var** *s* : sémaphore ; c_0 : entier)
- vérifier que $c_0 \geq 0$;
- *s.c* := c_0 ;
- *s.f* := {} ;

Agir sur les sémaphores

Pour un sémaphore donné, les deux procédures ci-dessous s'exécutent en exclusion mutuelle.

procédure P(**var** *s* : sémaphore)
- *s.c* := *s.c* − 1
- **si** (*s.c* < 0) **alors**
  - ⟨soit P le processus appelant⟩
  - ⟨entrer le processus P dans la file *s.f*⟩
  - ⟨suspendre le processus P⟩
- **fin si**

procédure V(**var** *s* : sémaphore)
- *s.c* := *s.c* + 1
- **si** (*s.c* ≤ 0) **alors**
  - ⟨sortir un processus Q de la file *s.f*⟩
  - ⟨reprendre le processus Q⟩
- **fin si**

P pour *proberen* (tester) ou *wait*

V pour *verhogen* (incrémenter) ou *signal*

Utilisation des sémaphores

Soient trois processus qui exploitent la même ressource :

	P_1	P_2	P_3
(1)	P(s)	⋮	⋮
(2)	⋮	P(s)	⋮
(3)	⋮	⋮	P(s)
(4)	V(s)	⋮	⋮
(5)	⋮	V(s)	⋮
(6)	⋮	⋮	V(s)

La trace de l'exécution donne :

	action	(2,{})	P_1	P_2	P_3
(1)	P(s)	(1,{})	SC	A	A
(2)	P(s)	(0,{})	SC	SC	A
(3)	P(s)	(-1,{P_3})	SC	SC	S
(4)	V(s)	(0,{})	A	SC	SC
(5)	V(s)	(1,{})	A	A	SC
(6)	V(s)	(2,{})	A	A	A

Les sémaphores simulés par des verrous

sémaphore : **structure**
- *blocage* : verrou ;
- *mutex* : verrou ;
- *c* : entier ;

procédure init(**var** *s* : sémaphore ; c_0 : entier)
- vérifier que $c_0 \geq 0$;
- *s.c* := c_0
- init(s.blocage) ;
- init(s.mutex) ;

Les sémaphores simulés par des verrous

procédure P(**var** *s* : sémaphore)

```
prendre(s.mutex) ;
s.c := s.c − 1
si (s.c = −1) alors
    libérer(s.mutex) ;
    prendre(s.blocage) ;
    prendre(s.blocage) ;
sinon si (s.c < 0) alors
    libérer(s.mutex) ;
    prendre(s.blocage) ;
sinon
    libérer(s.mutex) ;
fin si
```

procédure V(**var** *s* : sémaphore)

```
prendre(s.mutex) ;
    s.c := s.c + 1
    libérer(s.blocage) ;
libérer(s.mutex) ;
```


Les sémaphores privées

Soit *sempriv* un sémaphore privé de P_1 et *mutex* un sémaphore ordinaire :

P_1 :

```
P(mutex)
si ⟨le blocage est inutile⟩ alors
    V(sempriv)
fin si
V(mutex)
P(sempriv)
```

P_2 :

```
P(mutex)
si ⟨le processus P1 doit être débloqué⟩ alors
    V(sempriv)
fin si
V(mutex)
```


Le problème du producteur et du consommateur

Il existe un tampon de taille **limité** entre le producteur et le consommateur.

Algorithme du producteur :

```
répéter
    ⟨produire un message⟩
    ⟨le déposer dans le tampon⟩
jusqu'à ...
```

Algorithme du consommateur :

```
répéter
    ⟨prélever un message depuis le tampon⟩
    ⟨le consommer⟩
jusqu'à ...
```


Codage du producteur/consommateur

Le consommateur consomme si le tampon n'est pas vide :

NPlein : sémaphore = $(0, \{\})$

producteur :

```
répéter
    ⟨produire un message⟩
    ⟨le déposer dans le tampon⟩
    V(NPlein) ;
jusqu'à ...
```

Consommateur :

```
répéter
    P(NPlein) ;
    ⟨consommer⟩
jusqu'à ...
```

Codage du producteur/consommateur (2)

Définition des sémaphores :

NPlein : sémaphore $=$ $(0, \{\})$
NVide : sémaphore $=$ $(n, \{\})$

Le producteur produit si le tampon n'est pas plein :

répéter
  P(NVide) ;
  ⟨produire un message⟩
  ⟨le déposer dans le tampon⟩
  V(NPlein) ;
jusqu'à ...

Le consommateur consomme si le tampon n'est pas vide :

répéter
  P(NPlein) ;
  ⟨consommer⟩
  V(NVide) ;
jusqu'à ...

Les sémaphores à messages

Définition :

séma-mesg : **structure**
  c : entier ;
  demandeurs : file FIFO de processus ;
  messages : file FIFO de messages ;

Procédures :

procédure Pm(**var** s : séma-mesg ; **var** m : données)
  $s.c := s.c - 1$
  si $(s.c < 0)$ **alors**
    ⟨soit P le processus appelant⟩
    ⟨entrer le processus P dans la file s.demandeurs⟩
    ⟨suspendre le processus P⟩
  fin si
  ⟨sortir m de la file s.messages⟩

procédure Vm(**var** s : séma-mesg ; m : données)
  ⟨entrer m dans la file s.messages⟩
  $s.c := s.c + 1$
  si $(s.c \leq 0)$ **alors**
    ⟨sortir un processus Q de la file s.demandeurs⟩
    ⟨reprendre le processus Q⟩
  fin si

Les régions critiques

Soit une pile partagée par plusieurs threads :

pile = **structure partagée**
  sommet : **entier** ;
  data : **tableau** [1 .. Max] de **entier** ;

procédure empiler(**var** p : pile ; d : **entier**)
  région p **quand** (sommet < Max)
    sommet := sommet + 1 ;
    data[sommet] := d ;
  fin de région

procédure dépiler(**var** p : pile ; **var** d : **entier**)
  région p **quand** (sommet > 0)
    d := data[sommet] ;
    sommet := sommet - 1 ;
  fin de région

Les régions critiques simulées par sémaphores

L'instruction

```
région p quand (condition)
  | ⟨région critique⟩
fin de région
```

peut se coder par

```
var  mutex        : sémaphore = (1, {}) ;
     blocage      : sémaphore = (0, {}) ;
     nbEnAttente  : entier = (0) ;
```



```
P(mutex) ;
tant que non (condition)
  | nbEnAttente := nbEnAttente + 1 ;
  | V(mutex) ;
  | P(blocage) ;
  | P(mutex) ;

⟨région critique⟩

répéter nbEnAttente fois V(blocage) ;
nbEnAttente := 0 ;
V(mutex) ;
```

►► Gestion des ressources ◄◄

Allocation de ressources :

- Définition d'une ressource
- Objectif de l'allocation de ressources
- Le graphe de l'allocation de ressources

Les interblocages :

- Définition
- Comment traiter les interblocages ?
- Prévention des interblocages
- Évitement des interblocages
- L'algorithme des Banquiers
- Détection des interblocages
- Guérison des interblocages

Définition d'une ressource

Une *ressource* est un objet utilisable par une tâche. Elle est caractérisée par :

- un état (*libre* ou *allouée*),
- l'existence d'un *mode d'emploi*,
- l'existence d'un allocateur qui répond aux requêtes,

On distingue les ressources :

– *banalisées* qui ont des occurences multiples (imprimantes, canaux d'E/S),

– *réquisitionnables* (CPU, mémoire),

– *physiques* ou *logicielles*,

– *partageables* ou *réentrantes* (pour le code d'un programme).

Objectifs de l'allocation de ressources

Les allocateurs de ressources doivent :

– être *équitables* (en respectant les priorités),

– éviter la *privation* (attente sans fin d'une ressource),

– éviter l'apparition d'un *interblocage*,

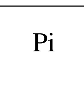
– éviter une *congestion* en veillant à :

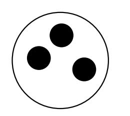
  - identifier une demande excessive de ressources,
  - ne pas accepter de demandes quand le système est en surcharge

Le graphe de l'allocation de ressources

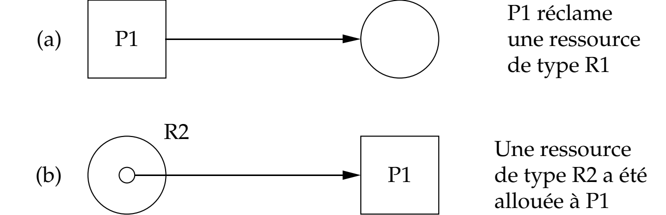
L'état des ressources allouées est décrit au moyen d'un *graphe de l'allocation de ressources*.

Un processus P_i et une ressource R_j ayant 3 occurrences sont décrits par :





Les demandes d'allocations et les ressources allouées sont décrits par :



Les interblocages

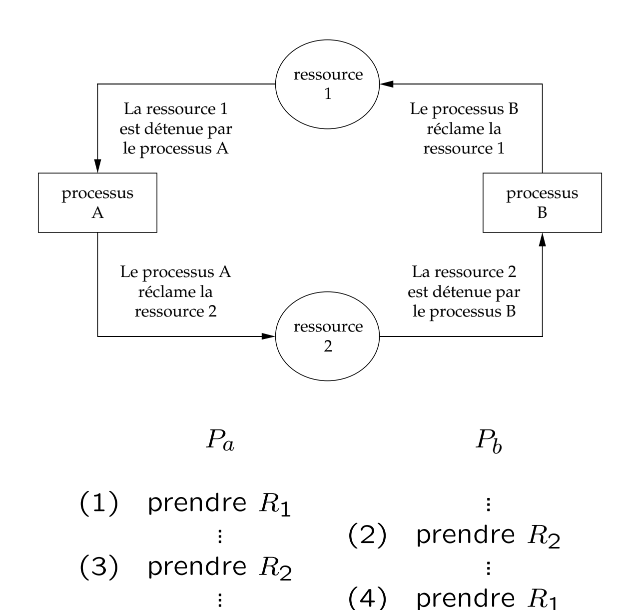
Définition : Attente **sans fin** pour plusieurs processus d'une ou plusieurs ressources détenues par d'autres processus.

Conditions d'apparition :

- les ressources ne sont pas partageables
- les ressources ne sont pas réquisitionnables
- chaque processus utilise simultanément plusieurs ressources
- il existe un ensemble de processus $(P_1, \ldots, P_n)$ tel que
  - P_i attend P_{i+1} pour $1 \le i \le n-1$
  - P_n attend P_1

Un exemple d'interblocage

Un interblocage représenté par un graphe d'allocation des ressources :



P_a	P_b
(1) prendre R_1	⋮
⋮	(2) prendre R_2
(3) prendre R_2	⋮
⋮	(4) prendre R_1

Comment traiter les interblocages ?

- La *politique de l'autruche.*
- La *Prévention* : On impose des règles strictes qui évitent l'apparition d'un interblocage.
- L'*évitement* : On surveille l'apparition d'un interblocage en imposant des contraintes moins strictes.
- La *détection* : Un algorithme à la charge de détecter les interblocages.
- La *guérison* : Suppression éventuelle de l'un des processus.

Prévention des interblocages

Diminuer le nombre de ressources qui ne peuvent être réquisitionnées.

→ interdire l'accès direct à la ressource,
→ remplacer la ressource **réelle** par une ressource **virtuelle**.

exemple

- processeur → processus,
- mémoire → mémoire virtuelle,
- imprimante → queue d'impression,
- écran → double-buffer, *display*,
- machine → machine virtuelle.

Prévention des interblocages

Contrainte 1 : Classer les ressources suivant un ordre, et respecter cet ordre lors des demandes de ressources.

+ La condition d'attente circulaire est impossible donc pas d'interblocage possible.
– Cet ordre doit respecter une certaine logique, ...mais laquelle ?
– La contrainte est très lourde...
– La portabilité des applications est médiocre.
– Que faire quand on ajoute une nouvelle ressource ?

Prévention des interblocages (suite)

Contrainte 2 : Annoncer les demandes de ressources avant de démarrer un processus.

Contrainte 3 : À chaque demande d'allocation d'une ressource supplémentaire, il faut libérer toutes les ressources détenues et les redemander en y ajoutant la ressource supplémentaire.

Conséquences :

+ solution parfaite,
– très mauvaise utilisation des ressources,
– programmes difficile à écrire.

Évitement des interblocages

On évite les interblocages en adoptant un comportement *prudent*. C'est l'*algorithme des banquiers*.

- le S.E. connaît les demandes maximales,
- les allocations/libérations sont libres.

Structures de données :

- M : nombre de classes de ressources,
- N : nombre de processus,
- dispo_j pour $i \in \{1, \ldots, M\}$,
- $\text{max}_{i,j}$ pour $i \in \{1, \ldots, N\}$, $j \in \{1, \ldots, M\}$,
- $\text{alloc}_{i,j}$ pour $i \in \{1, \ldots, N\}$, $j \in \{1, \ldots, M\}$,

L'algorithme des Banquiers

• Un processus P_i **peut s'exécuter** ssi

$$\forall j \in \{1, \ldots, M\}, \;\; (\text{max}_{i,j} - \text{alloc}_{i,j}) \leq \text{dispo}_j$$

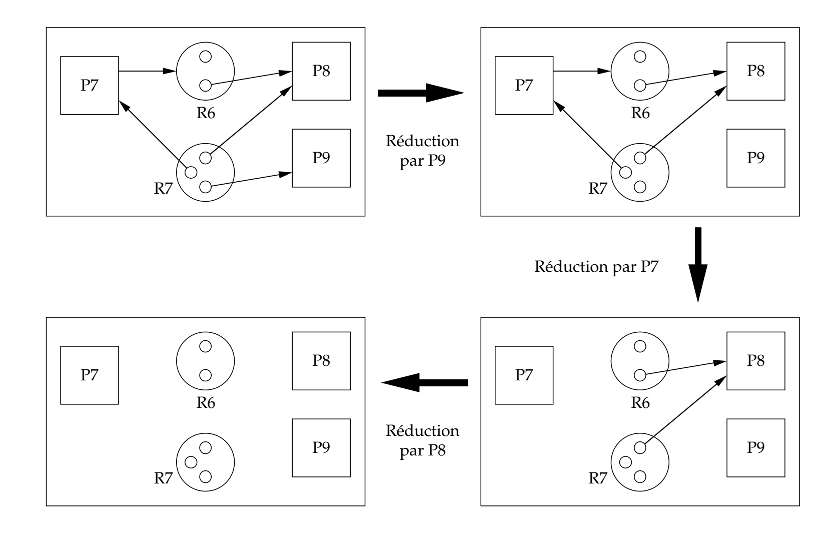
• Les indices $k_1, \ldots, k_n$ forment un **ordre d'exécution** si et seulement si les processus P_{k_i} (avec $i \in \{1, .., n\}$) peuvent s'exécuter dans cet ordre **les uns après les autres**.

• Un système d'allocation de ressources est dit **sain** si il existe un ordre d'exécution. Il n'y aura pas d'interblocages dans un système sain.

Algorithme d'allocation de R_j à P_i :

1. $\text{dispo}_j > 0$?
1. les annonces sont-elles respectées ?
2. si R_j est allouée à P_i, l'état est-il sain ?
3. si la réponse est négative, suspendre le processus P_i

Détection des interblocages



Détection des interblocages (suite)

Guérison des interblocages

Solution :

- Tuer un processus pour libérer des ressources. . .mais quel processus ?

Introduction des **transactions**.